AspectJ bajo la Lupa

Sandra Casas, Héctor Reinaga, y Cecilia Fuentes

Universidad Nacional de la Patagonia Austral (UARG)
Campus Universitario, Av. Gregores y Piloto Lero Rivera
C.P. 9400, Río Gallegos, Argentina
scasas@unpa.edu.ar

Abstract. La dependencia y acoplamiento existente entre aspectos y clases generan un alto impacto cuando se modifica el dominio de una aplicación. Esta situación es propia de aplicaciones que usan aspectos implementados en lenguajes del estilo AspectJ. En estos casos el desarrollador se enfrenta a nuevos dilemas puesto que el comportamiento introducido por los aspectos se ejecuta o deja de ejecutarse sin haber modificado dichos aspectos. La mejor manera de resolver un problema es anticiparse, se convierte en un principio y estrategia que acuñamos para encontrar una posible solución al problema planteado. En este camino, este trabajo propone una solución empírica basada en la herramienta BaLaLu, cuya finalidad es anticipar las consecuencias que un cambio podrá tener en una aplicación que usa aspectos.

Keywords: Programación Orientada a Aspectos, Separación de Concerns, Operaciones de Cambio, Evolución de Software, AspectJ.

1 Introducción

Las aplicaciones software están en permanente evolución. Las necesidades de dar soporte a nuevos requerimientos, adaptación a nuevos contextos, extensión de la funcionalidad son inherentes a toda clase de dominios. Esta realidad es independiente de metodologías, enfoques, herramientas y lenguajes de programación empleados en el desarrollo. Varios estudios [1] indican que los esfuerzos y costos de mantenimiento y evolución del software, superan ampliamente a los volcados en las etapas de desarrollo iniciales.

En estos escenarios de evolución continua, cambiar el código fuente permite modificar, adaptar, y extender el software. Aquí la noción de operación de cambio emerge con fuerza, tanto si es simple, como agregar un método a una clase, o mucho mas, cuando es compleja. Programar es más que editar texto, aboga Robbes [2], señalando que aunque la tarea trata con archivos y líneas de código es realmente ardua y problemática. El software asume variadas formas de abstracción, como clases, interfaces, aspectos, componentes, funciones, procedimientos, etc. y sobre estos operan diversas y múltiples relaciones e interacciones tales como herencia o composición que hacen complejo su comprensión para la modificación rápida y libre de errores. La falta de documentación y rotación de los desarrolladores hace más dura la tarea.

A esta realidad la Programación Orientada a Aspectos [3] (AOP) no escapa. Este enfoque surgió con la clara intención de proveer un mejor soporte a la separación de concerns [4]. La modularización de crosscutting concerns [5] en aspectos de busca evitar el código mezclado y disperso, y de ésta forma lograr que el software sea más mantenible, evolucionable y compresible. Sin embargo y contradictoriamente, sus mecanismos particulares de composición/invocación implícita la tornan susceptible a imperceptibles cambios que ocurran precisamente en los módulos funcionales. La dependencia y acoplamiento existente entre aspectos y clases generan un alto impacto cuando se modifica el dominio. Esta situación es propia de aplicaciones que usan aspectos implementados en lenguajes del estilo AspectJ [6]. En estos casos el desarrollador se enfrenta a nuevos dilemas puesto que el comportamiento introducido por los aspectos se ejecuta o deja de ejecutarse sin haber modificado dichos aspectos.

La mejor manera de resolver un problema es anticiparse, se convierte en un principio y estrategia que acuñamos para encontrar una posible solución al problema planteado. En este camino, este trabajo propone una solución empírica basada en la herramienta BaLaLu, cuya finalidad es anticipar las consecuencias que un cambio podrá tener en una aplicación que usa aspectos.

2 AOP y la Evolución

Un aspecto es la abstracción central de la AOP, el cual se compone de dos partes modulares: los pointcuts y los advices. Mientras que los advices son similares a los métodos, en cuanto a que son fragmentos de código que serán incorporados al dominio en algún momento de la ejecución (after-before-around); los pointcuts son las expresiones que establecen ante que eventos y condiciones estos fragmentos de código serán ejecutados, por ejemplo cuando ocurra la llamada de un determinado método. En los pointcuts es donde se encuentra el punto más crítico, ya que ante un mínimo cambio en el dominio, pueden alterar el conjunto de eventos y condiciones objetivo al momento de diseñarse e implementarse. Esto se debe principalmente, al hecho de que existe un alto acoplamiento y dependencia entre estas expresiones de pointcuts y el dominio. Debido a que un pointcut concreto referencia “explícitamente” los eventos del domino de interés, a la vez que lo referencia simbólicamente, a través del uso de comodines.

Entonces decimos que ocurre un *falso positivo* cuando un pointcut que interceptaba n join-points luego de una operación de cambio, intercepta m join-points, donde $m>n$. De igual manera, ocurre un *falso negativo* cuando un pointcut que interceptaba n join-points luego de una operación de cambio intercepta m join-points, donde $m<n$.

Las operaciones de cambio representan la evolución del software. Son las acciones que llevan adelante los desarrolladores cuando modifican el código fuente. Las operaciones de cambio representan la transición desde un estado de la evolución de una aplicación al próximo. Ejemplos de operaciones de cambio son: agregar una clase, renombrar un método, o aplicar algún refactoring [7]. Las operaciones de cambio se clasifican en “atómicas” y “compuestas”. Las operaciones atómicas implican una única acción indivisible. Las operaciones de cambio compuestas

resultan ser una secuencia de operaciones de cambio atómicas. Importan las operaciones de cambio en tanto son potencialmente generadoras de falsos positivos y/o falsos negativos en las aplicaciones que usan aspectos, y por ende su identificación resulta imprescindible.

En la Fig. 1 el aspecto LogCambiosPosicion, implementa el mecanismo de log para registrar los cambios de posición de los objetos de tipo Punto y Linea. La Tabla inferior indica como cambios muy simples en el dominio (clases Punto y Linea) tienen consecuencias en términos de falsos positivos y/o negativos.

```
public aspect LogCambiosPosicion {
   pointcut cambioPosicionPunto(Punto p): call(* Punto.set*(int))&&
                                          target(p);
   after(Punto p): cambioPosicionPunto(p) {
      Logger.writeLog("Cambio posición Punto: "+p.toString());
   }
   pointcut cambioPosicionLinea(Linea l): call( * Linea.set*(Punto))
                                          && target(l);
   after(Linea l): cambioPosicionLinea(l) {
      Logger.writeLog("Cambio posición Linea: "+l.toString());
   }
}
```

CAMBIO	CONSECUENCIA
1) Renombrar el tipo Punto por MiPunto	la intercepción de {call (void Punto.set*(int) } es vacío y se producen falsos negativos
2) Cambiar la signatura de void setX(int) por void setX(double) en la clase Punto	{call (void Punto.set*(int)} se rompe y se producen falsos negativos.
3) Agregar un atributo a las clases Punto y/o Linea, que no refiera a la posición y luego agregar el método set respectivo para modificar su estado.	las llamadas a este método serán interceptadas por alguno de los pointcuts del aspecto LogCambiosPosicion, y se producen falsos positivos

Fig. 1. Aspecto LogCambiosPosicion.

El problema descripto se conoce como “fragile pointcuts” [8] y representa un verdadero problema para la evolución del software que usa aspectos. Otros inconvenientes relacionados para la evolución de aplicaciones con aspectos son abordados en [9] [10] [11] [12].

3 Operaciones de Cambio

Desde nuestra perspectiva, una operación de cambio es una *función* que se aplica a una representación del código fuente, con el objeto de conocer anticipadamente sus potenciales consecuencias. Independientemente de la naturaleza del mantenimiento (correctivo, adaptativo, perfectivo, preventivo) se sabe que un conjunto de operaciones de cambio tendrá lugar en el código fuente, lo que conformará una nueva versión de la aplicación software. La cantidad de operaciones de cambio aplicadas en cada versión, dependerá de múltiples factores como cantidad y tipo de requerimientos, experiencia del desarrollador, lenguaje de programación, etc.

Desde el enfoque, se propone que la operación de cambio se aplicará al repositorio R_n. Las distintas instancias de $R = \{R_1, R_2, \ldots, R_n\}$ se presentan coincidentemente con las versiones de programa, $V = \{V_1, V_2, \ldots, V_n\}$. Esto produce que una determinada operación si es aplicada a diferentes instancias de R, pueda generar distintas consecuencias.

Al definir una operación de cambio como una función, es necesario especificar en cada caso, el conjunto de entrada y de salida. En el caso de las operaciones de cambio compuestas, importa identificar la secuencia de operaciones de cambio que las componen en su preciso orden de aplicación. Lo que posibilita que una operación de cambio compuesta sea una secuencia de operaciones de cambio atómicas y/o compuestas y sus entradas y salidas están determinadas por las operaciones de cambio que pertenecen al conjunto definido. De modo general, informalmente se especifican las operaciones de cambio:

```
<operación_de_cambio> ::= <operación_de_cambio_atomica> |
                          <operación_de_cambio_compuesta>
                          → <consecuencias>
<operación_de_cambio_atomica> :: <entradas> → <consecuencias>
<operación_de_cambio_compuesta> ::= {<operación_de_cambio_atomica> |
                                    <operación_de_cambio_compuesta>}
                                     → <consecuencias>
```

Operaciones de cambio atómicas identificadas:

```
removePackage :: identificador_package → falsos_negativos
removeClass :: identificador_class → falsos_negativos
removeMethod :: identificador_method, from_class → falsos_negativos
removeField :: identificador_field, from_class → falsos_negativos
removeMessage :: identificador_message, from_method, from_class →
falsos_negativos
removeHandler :: identificador_exception, from_method, from_class →
falsos_negativos

addPackage :: identificador_package → falsos_positivos
addClass :: identificador_class → falsos_positivos
addMethod :: identificador_method, to_class → falsos_positivos
addField :: identificador_field, to_class → falsos_positivos
addMessage :: identificador_message, to_method, to_class →
falsos_positivos
addHandler :: identificador_exception, to_method, to_class →
falsos_positivos
```

Operaciones de cambio compuestas identificadas:

```
moveClass(id_class from_pack to_pack) :: removeClass(id_class
from_pack), addClass(id_class to_pack) → falsos_positivos,
falsos_negativos

moveMethod(id_method from_class to_class) :: removeMethod(id_method
from_class), addMethod(id_method to_class) → falsos_positivos,
falsos_negativos

moveField(id_field from_class to_class) :: removeField(id_field
from_class), addField(id_field to_class) → falsos_positivos,
falsos_negativos

moveMessage(id_message from_method to_message) ::
removeMessage(id_message from_method), addMessage(id_message
to_methodclass) → falsos_positivos, falsos_negativos
```

Siguiendo el esquema, se han especificado las funciones para la operación de cambio “rename”, aplicada a las distintas entidades: package, class, method, field.

4 BaLaLu: anticipando los cambios

BaLaLu (Bajo La Lupa) es una herramienta que hemos diseñado para anticipar las consecuencias ante una operación de cambio en el dominio. Las operaciones de cambio simples y compuestas que se han implementado, son las referidas en la Sección anterior. Un repositorio administra el modelo de datos de manera tal que representa a los programas (clases, aspectos, interfaces, etc.) de una aplicación que usa aspectos. Las operaciones de cambio son introducidas a través de una plantilla general que permiten simplemente configurar los parámetros de las funciones.

En el repositorio el código fuente es representado en términos de entidades, elementos, relaciones e interacciones, como se grafica en la Fig. 2. En esta representación, la entidad Pointcut asume una relevancia particular dado que son exhaustivamente analizadas ante la ejecución de cualquier función de operaciones de cambio.

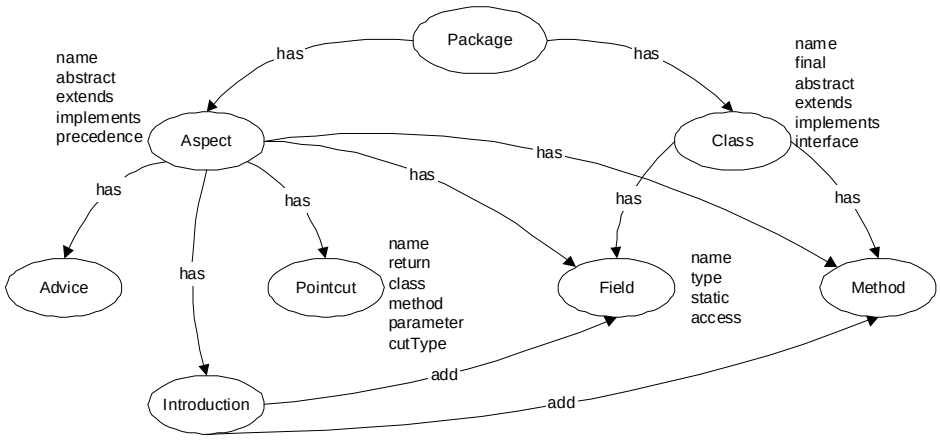

Fig. 2. Entidades principales del repositorio de BaLaLu.

Las operaciones de cambio son introducidas de una manera muy intuitiva. BaLaLu consta de un formulario (Fig. 3) que permite seleccionar los elementos del dominio requeridos como entrada de la función que se pretende aplicar. Esta se configura particularmente con los elementos de cada aplicación.

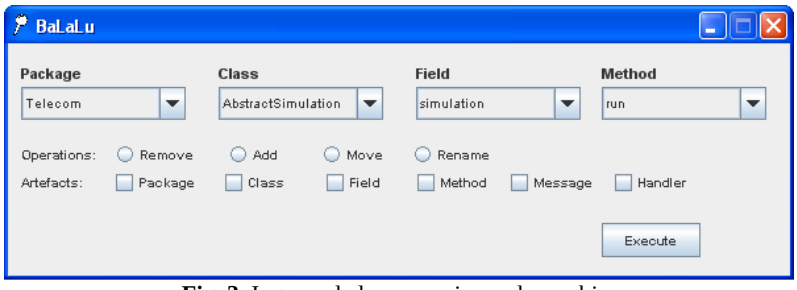

Fig. 3. Ingreso de las operaciones de cambio.

Las consecuencias de una operación de cambio generan un reporte que indica la cantidad de falsos positivos y/o negativos que se han producido, y el detalle de dónde se producen los mismos (aspecto, pointcut, expresión) como se presenta en la Fig. 4.

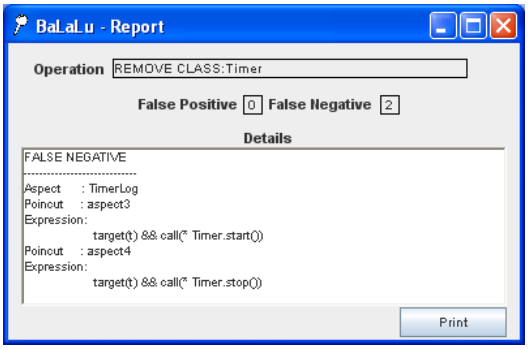

Fig. 4. Reporte de consecuencias.

4 Ejemplo: Telecom

Telecom es una simple aplicación distribuida con AspectJ, que simula conexiones telefónicas. La aplicación tiene tres funcionalidades principales: Customers, Calls y Connections. Las características de las clases se describen a continuación:

- Call: Representa una llamada, crea una conexión de tipo local o larga distancia.
- Connection: Representa una conexión entre los clientes, y lo modela a través de una máquina de estados simples (las conexiones son inicialmente PENDING, luego COMPLETED y finalmente DROPPED). Los mensajes se imprimen en la consola de manera que el estado de las conexiones pueden ser observadas. La clase Connection es abstracta con dos subclases: Local y LongDistance.
- Costumer: Representa un cliente, tiene métodos de call(), pickup(), hangup() y merge() para administrar las llamadas.
- Timer: Representa un contador de tiempo, y registra los tiempos cuando se inicia y se detiene una conexión; así también, devuelve la diferencia del tiempo transcurrido.

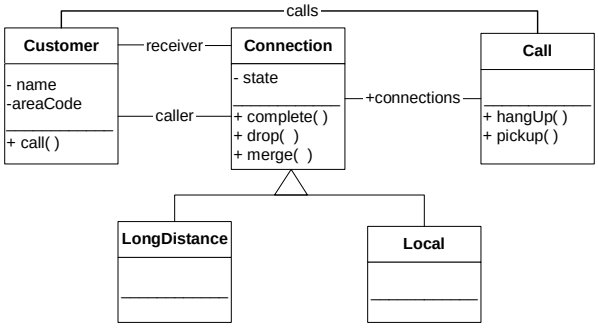

Fig. 5. Diagrama de Telecom.

También incluye tres aspectos, los cuales implementan los requerimientos de tiempo y facturación de las conexiones, los que se describen a continuación:
- Timing: añade funcionalidad para registrar el tiempo de conexión telefónica utilizando un timer a cada conexión.
- Billing: calcula el coste de cada conexión y se basa en la información de tiempo obtenida del aspect Timing.
- TimerLog: guarda en un log el registro de cuando empieza y termina un timer, por lo tanto, este aspecto intercepta la ejecución de los métodos Timer.start() y Timer.stop().

La aplicación descripta ha sido representada en el repositorio de BaLaLu, y se han aplicado las siguientes operaciones de cambio: añadir un tipo de conexión (Fig. 6), renombrar la clase Connection (Fig. 7), y renombrar el método start (Fig. 8).

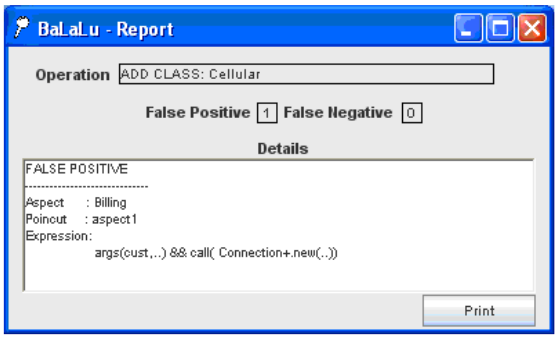

Fig. 6. Reporte operación Add Class Cellular.

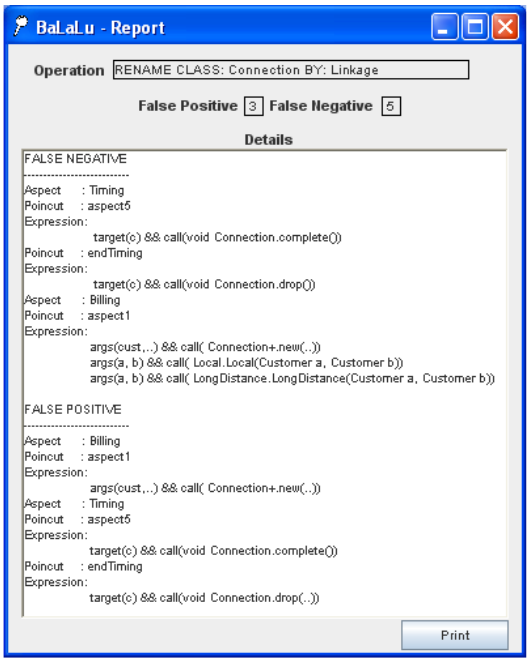

Fig. 7. Reporte operación Rename Class Connection.

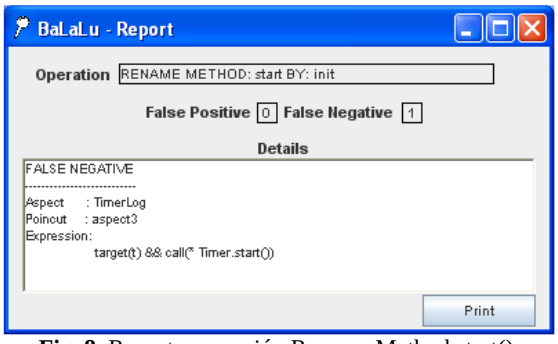

Fig. 8. Reporte operación Rename Method start().

En la Fig. 6, se añade un subtipo de clase de Connection, llamada Cellular, produciéndose falsos positivos, dado que intercepta las llamadas a {call (Connection+.new(..))}, correspondiente al pointcut aspect1, del aspecto Billing. La Fig. 7, renombra la clase Connection por Linkage, produciéndose falsos negativos y positivos, debido a que se trata una operación compuesta (aplica las operaciones Remove y Add Class), siendo interceptados por los aspectos Timing, endTiming, y Billing, para aquellos casos donde el designador de clase corresponda con la clase a renombrar. Por último, la Fig. 8, renombra el método start() de la clase Timer, por init(), lo cual produce falsos negativos, dado que la intersección de {call (* Timer.start())} es vacío y produce falsos negativos, corresponde al pointcut aspect3, del aspecto TimerLog.

5 Trabajos Relacionados.

SpyWare [2] [13, 14, 15] es un IDE que encarna el enfoque de evolución de software basado en cambios (CBSE). CBSE surge en contraposición y como superador de los típicos sistemas de configuración de versiones. CBSE trata a los cambios como entidades de primera clase. Una diferencia que se observa con nuestra propuesta es el propósito del modelo CBSE, el cual define la historia de un programa como la secuencia de cambios que el programa atravesó. Al disponer de la historia de cambios se puede reconstruir cada estado sucesivo del código fuente de los programas. A partir de esta diferencia que consideramos sustancial, encontramos otras como ser: mientras que para CBSE las operaciones de cambio se establece como entidades de primera clase, para la anticipación se definen como funciones; el éxito del modelo CBSE requiere que pueda ser implementado en los IDES o herramienta de desarrollo, mientras que el modelo de anticipación puede implantarse en estos IDES o en herramientas especificas. El trabajo abarco solo aplicaciones OO (Java y Squeak) y no fue considerada aún el uso de aspectos, aunque suponemos que la extensión del modelo a aspectos es pausible.

PCDiff [8] es un plugin de Eclipse que aborda una solución estática para el problema de fragile pointcut de AspectJ. La herramienta analiza y compara los conjuntos de join-point interceptados en dos versiones de programas distintas (versión

original y versión editada) y a partir de las diferencias de estos conjuntos se derivan un conjunto de cambios atómicos identificados. El objetivo de PCDiff coincide con el de BaLaLu en tanto pretende lograr la identificación de las consecuencias de un cambio y alertar a los programadores de los cambios en la intercepción del comportamiento de los advices. La principales diferencias radican en: a) se plantea un proceso que se base totalmente en la comparación de versiones de programas, lo que implica que el impacto se analiza y detecta a posterior del cambio, mientras que nosotros proponemos analizar los cambios sobre un repositorio que represente a los programas "antes" de efectuar los cambios en el código; b) dado que trabaja con versiones de programas, solo puede hallar diferencias y consecuencias en términos de operaciones atómicas, no pudiendo definir si estas devienen de operaciones compuestas como son los refactoring. Todas las operaciones de cambio atómicas halladas se ubican así al mismo nivel, resultando realmente complejo analizarlas de manera mas relacionadas y que esto además tenga mayor significado para el desarrollador.

Herramientas automatizadas como AJDT [16] y PointcutDoctor [17] ante una expresión de pointcut muestran los join-points efectivamente interceptados y también aquellos "casi" interceptados, lo cual resulta útil ante una operación de cambio en los aspectos, pero resulta insuficiente para las operaciones de cambio que tienen lugar en el dominio.

6 Conclusiones.

El objetivo de este trabajo es hacer las tareas de mantenimiento y evolución del software que usan aspectos, mas predecibles y menos costosas. Mediante la representación de las entidades que conforman el código fuente de una aplicación, y un conjunto de operaciones de cambio inicialmente identificadas, se ha demostrado que es posible anticipar y/o predecir las consecuencias de un cambio en el dominio. De esta forma, es posible evitar que un efecto no deseado deba ser rastreado mediante intensas pruebas e inspecciones de código manual.

Nuestro trabajo actual se enfoca a dar implementación a operaciones más complejas como refactoring y a las operaciones de cambios que sucedan en los aspectos. También nos encontramos abocados en el desarrollo de una versión de BaLaLu como plugin de Eclipse.

Referencias.

1. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. IEEE IT Pro, pp. 17-23, May (2000)
2. Robbes, R., Lanza, M.: An Approach to Software Evolution Based on Semantic Change. In: Proceedings of Fase 2007, pp. 27, 41 (2007)
3. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer-Verlag (1997)
4. Dijkstra, E.W.: "A Discipline of Programming". Prentice-Hall. (1976)

5. Hürsch, W., Lopes, C.: “Separation of Concerns”. Northeastern University Technical Report NU-CCS-95-03, Boston. (1995)
6. Kiczales, G.: Tutorial on Aspect-Oriented Programming with AspectJ, FSE (2000)
7. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
8. Koppen, C., Stoerzer, M.: Pcdiff: Attacking the Fragile Pointcut Problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany (2004)
9. Coelho, R., Rashid, A., Garcia, A., Ferrari, F., Cacho, N., Kulesza, U., Staa, A., Lucena, C.: Assessing the Impact of Aspects on Exception Owns: An Exploratory Study. In: European Conference on Object-Oriented Programming (ECOOP), pp. 207-234 (2008)
10. Soares, S., Borba, P., Laureano, E.: Distribution and Persistence as Aspects. Software: Practice&Experience., Vol. 36 (7): pp. 711-759 (2006)
11. Figueiredo, E., Cacho, N., Sant’Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Filho, F. C., Dantas, F.: Evolving Software Product Lines with Aspects: an Empirical Study on Design Stability. In: ICSE 08: Proceedings of the 30th International Conference on Software Engineering, pp. 261-270, New York, NY, USA (2008)
12. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In: European Conference on Object-Oriented Programming (ECOOP), number 4067 in LNCS, pp. 501-525 (2006)
13. Robbes, R., Lanza, M.: Change-Based Software Evolution. EVOL 2006, pp. 159- 164 (2006)
14. Robbes, R., Lanza, M.: A Change-Based Approach to Software Evolution. In: ENTCS, volume 166, issue 1, pp. 93-109 (2007)
15. Robbes, R., Lanza, M.: Towards Change-Aware Development Tools. Technical Report at USI, 25 pages (2007)
16. AJDT: AspectJ Development Tools, http://www.eclipse.org/ajdt/
17. Ye L., De Volder, K.: Tool support for understanding and diagnosing pointcut expressions. In: International Conference Aspect-Oriented Software Development (2008)